Introducing SERCs Safer Erlang

Now Superceeded by SSErl - Prototype of a Safer Erlang

Dr Lawrie Brown
School of Computer Science, Australian Defence Force Academy, Canberra, Australia
Email: Lawrie.Brown@adfa.edu.au

Last updated: 24 April 1997.

Abstract

In order to support outsourced and third party telecommunications applications, there is a desire to modify the Erlang language and execution environment to provide safe and partitioned execution of externally sourced or outsourced programs which are imported and run on a local Erlang system. This paper outlines a possible design approach, and describes the initial prototype.

Introduction

Erlang is a declarative language for programming concurrent and distributed systems which was developed at the Ericsson and Ellemtel Computer Science Laboratories [AVWW96], [Arms96], [Wiks94]. It is a dynamically typed, single assignment language which uses pattern matching for variable binding and function selection, has explicit mechanisms to create concurrent and distributed processes, and advanced facilities for error detection and recovery.

Mobile Code is code sourced from remote, possibly "untrusted" systems or suppliers, but imported and executed on a local system. Consequently such code needs to be executed within some form of "constrained" or "sandbox" environment to protect the local system from accidental or deliberate inappropriate behaviour.

Given the anticipated rapid growth in telecommunications applications software, there is expected to be a rapidly increasing need to support third party outsourced code being executed on trusted systems. It is believed this can be done with an acceptable level of safety by the use of containment methods being developed to support the concept of mobile code, as used in Java [GM95], SafeTCL [OLW96], Omniware [LSW95], and Telescript [Tar95], amongst other systems (see overview in Brow96b).

The approach being considered here is support the execution of a number of mutually untrusted (and untrusting programs) within a dedicated Erlang node. This involves partitioning the node into a collection of separate "subnodes", which provide a restricted execution environment or "sandbox", along with controlled access to processes in other "sandbox" environments. Each of these "sandboxes" form a separate security domain, where the operations available will be constrained by an appropriately chosen security policy. Different policies can be enforced in different "sandboxes".

Making Erlang Safer

The Erlang language provides a number of inherent benefits. Its dynamic typing and single assignment prevent many classes of errors. The main additions involve controlling access to resources used to create and communicate with other processes, and to external devices. Currently this is through the use of Pids and Ports, with few restrictions on their use. Also, there is a need to partition a single "hardware" erlang node into a number of subnodes, each with a custom view of the world (in terms of registered names and which modules are available and used).

I propose protecting the former (Pids and Ports), as well as Nodes, by making them **password capabilities** [APW86]. In a password capability system, the capability is a data item which indicates the entity owning it (a node in this case), and a random value (selected sparsely from a large address space). An appropriate capability must be supplied, explicitly or implicitly, in order to perform most "unsafe" operations (which in Erlang involve the use of BuiltIn Functions - BIFs). The user is free to try and forge a capability, but it is statistically highly improbable that they’ll create a valid one. The capability has no meaning on its own, but is only of use when supplied to its owner (a node) along with a request for some operation. One advantage of password capabilities is the ease of revocation, by removing it from its owning entity’s list of currently valid capabilities. Any process subsequently trying to use it will fail with an invalid_capability (as also occurs if a forged capability is used).

For the latter, I propose creating a concept of **subnodes** to provide custom views. Each subnode should provide a "context" for processes executing in it. It provides distinct "registered names" and "module alias" tables. The registered names can be modified by processes with an appropriate subnode capability, and is used to send messages to named servers. The "module alias" table is specified when the node is created, and is used to alias module names at run-time when functions are invoked. By appropriate customisation, modules executing within a subnode can be provided with a custom view of modules used and servers available.

Prior Work

A system with similar goals, though with a stronger focus on code mobility, was SafeErlang, developed by Gustaf Naeser et al. at Uppsala [Nae97a], [JNS97].

SafeErlang incorporates three new concepts. **Encrypted capabilities** are used for pids and nodes, but not (yet) for open_port. This seriously limits its ability to protect key standard library routines, and accesses to external resources. Also the use of encryption introduces problems with appropriate key distribution in distributed environments, along with the selection of appropriate cryptographic algorithms. **Subnodes** are used to provide a custom collection of modules (with names rewritten at compile time), and to provide resource limits on processes executing in the subnode. Lastly, a new **module loading** (mid) mechanism is supplied to support code mobility. A new "code" server is used to manage the code (mid) distribution as required. In their current system, this necessitates a recompilation of source every time the module is loaded (in part to cope with varying module names needed, depending on which subnode it is being loaded in to).

The focus of their project was to support mobile agents, reflected in the emphasis on providing new module management functions, and providing resource limits for subnodes. For this, as reported in [JNS97], it has been successful. However I found the arrangement of servers used, and the division of responsibility for implementing various operations, to be unnecessarily complex. In SafeErlang, each "real" erlang node (system) has a "code" server which manages the new module loading mechanism; a "gate" server which manages the keys for all the subnodes on that system; and a "name" server which implements the replacement registered name system. Also for each (sub)node on the system, there is a "node" server process which manages the information for that subnode. Every time an "unsafe" BIF is invoked in a user process, a request is sent to the "gate" server which decrypts and checks the capability supplied. Usually a message is then sent from the "gate" server to the relevant node manager which actually implements the requested operation. Consequently the node managers must explicitly manage all the requested links between processes. Responses are returned via the same path. I believe this involves an unnecessary amount of message passing and state maintenance, and that a simpler and cleaner design is possible.

SERCs Safer Erlang Prototype

SERCs Safer Erlang (SSErl) is a prototypical implementation of what I believe to be a simpler and more comprehensive design of a secure erlang execution environment. It is less concerned with module mobility (though that should be added later by providing a custom error handler along with the use of the module alias mechanism), than with providing a comprehensive and elegant implementation of capabilities and subnodes with as few changes as possible to the existing erlang language specification. It is currently implemented as a collection of glue functions substituted by a modified erlang compiler for all calls of "unsafe" BIFs. These interact with "node" server processes, one for each distinct (sub)node on the erlang system. The prototype supports capabilities for pids, ports, and nodes; and a hierarchy of subnodes on each erlang system. Its uses a modified compiler (adapted from that developed by Naeser [Nae97a] for the SafeErlang system).

Most of the glue functions have the form:

- check with the node manager to see if the operation is permitted by the capability
- if so some key information is returned (generally a pid or a capability)
- perform the desired operation (in the users process) if necessary

Some routines require two capability checks (generally to see if the executing process is permitted to do, and the target capability permits, the desired operation). Functions to spawn new processes, open a port, or create a new subnode are also a little, but not much, more complex, since they must initialise some data structures. I believe this general structure mirrors fairly closely the logic that should be followed if these features were to be implemented within the Erlang RunTime System (ERTS) for production use.

Each SSErl process maintains a record of information which includes:

self
: a capability for itself, which determines which (potentially unsafe) operations the process is permitted to perform.

parent
: a capability for its parent node, used to restrict rights for newly created subnodes, and some other operations.

group_leader
: a capability for the process group_leader (used for I/O).

apply check function
: the name of the function which may (optionally) be called by the apply glue routine before any external function calls to validate those calls, described in [Bro97b]. Copied from the parent node state table.

aliases

the list of module aliases, used redirect external function calls from "well-known" library modules to safer variants. Copied from the parent node state table.

This record is stored in the process dictionary, and is protected by modified put and erase functions from changes outside the sserl glue routines.

Capabilities

In the prototype, a capability is a tuple {Type,Node,Ref,Rand}, where Type is one of capapid|capaport|capanode|caparef; Node is the name of the (sub)node which created the capability; Ref is an erlang reference value; and Rand is a random number. The latter two provide the statistical protection for the capability. The capability has no intrinsic meaning until it is supplied to the owning node manager. The node manager maintains as part of its state a list of [{Capa,Value}*] which maps a capability to its value. Value is a tuple of the form {Class,Val,Rights}, where Class identifies the object referenced as one of process|port|node|{user type}; Val is the process id, port no, node manager process id, or user supplied value (atom or integer); and Rights is the list of access rights. Currently the access rights supported for the various classes are:

process	db, exit, garbage_collect, group_leader, kill, link, local, open_port, priority, process_info, register, restrict, revoke, send, spawn, spawn_link, trace, trap_exit, unlink, unregister, view
port	link, local, restrict, revoke, send, unlink, view
node	delete_module, info, link, load_module, local, net_kernel, newnode, processes, restrict, revoke, shutdown, spawn, spawn_link, unlink, view
ref	local, restrict, revoke, view

Most of these correspond to permitting the BIF of the same name (or the process_flag for trap_exit or priority). Rights specific to capability manipulation include:

info	permits access to node state information;
local	indicates that name registrations may not be made globally available;
restrict	permits restriction of the capability
revoke	permits revokation of the capability (provided it is a restricted variant).
view	permits viewing of the capability value {Class,Val,Rights}.

An appropriate capability must be supplied either explicitly (as a pid/port/node argument), or implicitly (from the processes knowledge of its own capability, or its parent node's capability), in order to perform most "unsafe" BIFs.

A capability is created whenever a process is spawned, a port is opened, a subnode is created, a reference is made, or an existing capability is restricted. They may also be created with very limited rights for existing processes outside the SSErl environment as part of its initialisation, or to correspond to pids from a list_to_pid BIF. Capabilities are destroyed (removed from the relevant node table) when the associated object (process, port or node) dies.

User capabilities (references with a user supplied type and value) are intended to assist in providing finer control for file accesses, I/O device accesses, or other potentially sensitive operations.

Subnodes

SSErl subnodes provide a distinct context for processes executing within them. Each subnode has a "node" server process which maintains the state for that subnode. The servers are registered by their node name in the "real" erlang system registered names table. This allows glue functions executing in user processes to communicate with a specified node server (as given by the node name embedded in a capability). This also allows access to non-local node servers by sending a message to '{node host}'.

The state managed by the server process for each subnode includes:

name
: the name of the (sub)node as an atom, extended from the system name

self
: a capability for itself (defined in its own capability table).

parent
: a capability for its parent (defined in its parents capability table)

capability table [{Capa,Value}*]
: maps capabilities to their associated real data (pid) & rights

registered name table [{Name,Capa,Pid}*]
: maps names to a process capability, thus permitting different subnodes to have the same name referencing different processes, allowing custom variants of standard services to be provided

module alias table [{Name,Alias}*]
: remaps module names at runtime to an alias name, allowing different subnodes to direct the same module name to different actual modules. Currently this uses an exact match, it may be extended to include a prefix-match, allowing name extensions on all otherwise unmatched names, if desired.

subnode table [{CNode,Pid}*]
: provides a list of all subnodes which are children of this subnode

process table [{CPid,Pid}*]
: provides a list of all processes belonging to the subnode

prototypical process rights
: used to restrict rights for newly created processes in the node

apply check function {Mod,Func}
: the name of the function which may (optionally) be called by the apply glue routine before any external function calls to validate those calls, see [Bro97b].

Glue Functions

Glue functions have been provided for a number of "unsafe" BIFs in order to implement the capability and subnode functionality, and to impose more stringent checks on the right to perform these functions. These "unsafe" BIFs may be catagorised in groups as: Apply, Spawn, Module, Node, Process, Misc and Db. There are also some new BIFs to manipulate capabilities and subnodes. Calls made to these BIFs are replaced by the modified compiler (more specifically by the sys_pre_expand module), eg. erlang:spawn(M,F,A) becomes sserl_bifs:k_spawn(M,F,A).

As mentioned earlier, most of the glue functions have the form:

- check with the node manager to see if the operation is permitted by the capability
- if so some key information is returned (generally a pid or a capability)
- perform the desired operation (in the users process) if necessary

For example, the `link` glue routine is:

```
    k_link(CPid) ->
        Pid = node_request(check,CPid,link),
        link(Pid).
```

By category, the glue functions are:

Apply

apply({Module,Function}, ArgList)
apply(Module, Function, ArgList)
: both safe and unsafe calls could result from a call to apply. Nested applies are checked, safe variants are allowed, unsafe variants are rewritten, the apply check function is called (if specified), the module names are aliased, and finally the desired function is called. A process capability is returned.

Spawns and Open_Port

spawn(Module, Function, ArgList)
spawn_link(Module, Function, ArgList)
: these check that the requesting process has the right to, and the parent node permits, the spawn to occur. The new process is created and initialised. A process capability for the new process is returned.

spawn(Node, Module, Function, ArgList)
spawn_link(Node, Module, Function, ArgList)
: these spawn the process on a separate subnode (on either the same or a different underlying erlang system), returning a process capability.

open_port(PortName, PortSettings)
: creates a new port, and returns a capability for it, if the process is permitted. Unfortunately this affects communications with the port, since the real underlying Pid is used as part of the communications. Currently there is no easy way to catch and rewrite this code, so port interaction code must be rewritten slightly to work with the SSErl system.

Module

delete_module(Module)
: checks the process is permitted to, and aliases the module name before deleting the module.

load_module(Module)
: checks the process is permitted to, and aliases the module name before loading the module.

purge_module(Module)
: checks the process is permitted to, and aliases the module name before purging the module.

check_process_code(Pid, Module)
: retrieves the real pid, aliases the module name and is permitted.

function_exported(A1,A2,A3)
: is permitted as is.

module_loaded(Module)
: aliases the module name and is permitted

pre_loaded()
: is permitted as is.

Node

alive(Name, Port)
    blocked as redundant
disconnect_node(Node)
    blocked as redundant
get_cookie()
    permitted, but should become redundant
set_cookie(Node, Cookie)
    permitted if self has net_kernel right, though should become redundant
is_alive()
    permitted, but redundant
monitor_node(Node, Flag)
    permitted if self has net_kernel right
newnode(Name)
newnode(Parent,Name)
newnode(Parent,Name,{Node_Rights,Proto_Rights,Names,Aliases,Options,Apply_Chk})
    new BIFs to create a subnode of the parent node, if processes parent node capability permits.
    Any option values not supplied (given as nil) will be inherited from the parent node.
node()
    returns the name of the parent node of the process (should really be a capability)
node(Arg)
    returns the name of the node which created Arg (should really be a capability)
node_info(Node)
    new BIF to return the node state information if Node permits info.
nodes()
    returns a list of node names (should really be capabilities)
halt()
shutdown(Node)
    new BIF to terminate a node (halt defaults to parent node) if shutdown is permitted. If the
    topnode on an erlang system is shutdown, then the entire system is halted.

Process Communications

exit(Pid, Reason)
    used to cause a process to exit if permitted
kill(Pid)
    used to kill a process if permitted (equiv to exit(Pid,kill) but with a separate right)
group_leader()
    returns the capability of the group leader process
group_leader(Leader, Pid)
    used to set the group leader of a process if permitted
link(Pid)
    links to a process if permitted
list_to_pid(List)
    converts a list to a pid, returning a (very restricted) capability
pid_to_list(Pid)
    converts the pid referenced by the capability to a list if view permitted
processes()
processes(Node)
    returns a list of all process capabilities executing on a node (defaults to parent node).

process_flag(Flag, Option)
: used to set the trap_exit or priority flags, if self has the corresponding right. Altering the error_handler is not permitted.

process_info(Pid)
process_info(Pid, Key)
: retrieves process information if permitted.

register(Name, Pid)
: used to register a name and corresponding capability on the parent node if both the capability and self permit. Any form of capability may be registered. Currently names are strictly local. A global server may be supported later.

registered()
: retrieves a list of registered name and capability pairs.

self()
: returns the processes own capability.

send(To,Msg)
: BIF which handles the message send operation " To!Msg". "To" may be a locally registered name, a remotely registered name and remote node capability, or a process capability; which identifies the process to receive the message.

unlink(Pid)
: unlinks from a process if permitted.

unregister(Name)
unregister(Pid)
: unregister the name or corresponding pid if it permits.

whereis(Name)
: returns the capability referenced by the registered name on the parent node.

Capability Functions

check(Capa,Op)
: new BIF to check if the capability permits op.

make_ref()
make_ref(Type,Val)
: creates a reference capability. The latter call associates a user Type (an atom) and Val (an atom or integer) with the reference capability for use as a user capability.

restrict(Rights)
: restricts the list of rights for a processes self capability - does not create a new capability. nb. the new list of rights will be the intersection of the existing and supplied lists of rights.

restrict(Capa,Rights)
: new BIF to create a restricted version of an existing capability. nb. the new list of rights will be the intersection of the existing and supplied lists of rights.

revoke(Capa)
revoke(Capa,Master)
: new BIF to revoke a capability restricted from self or Master.

same(C1,C2)
: new BIF to test whether the two capabilities (which may be restricted variants) refer to the same underlying object (eg pid or port).

view(Capa)
: new BIF to access the information the capability refers to (its real pid and access rights) if view is permitted.

Misc

erase()
put(Key,Value)
    permitted except that the sserl process information may not be erased or modified.
garbage_collect()
garbage_collect(Pid)
    permitted if self or pid respectively permits garbage_collect.
statistics(Type)
    permitted.
trace(Pid,How,Flags)
    permitted if pid permits trace.

DB

All the various "db_*" BIFs are replaced by functions which check self permits db before they are called. This probably ought to be refined further, but has not been a priority for this prototype.

Using the SSErl Prototype

The SSErl prototype is distributed a tar file which includes source and precompiled jam files for the sserl, modified compiler, and changed standard library modules. A README file is included which describes the (minimal) customisation required. Also required is a working Erlang 4.4.1 system (available from [Erla96]).

Once installed, a Unix shell script `sserl` is used to start the SSErl system. It is invoked as `"sserl"` (normally), or `"sserl -verbose"` (for copious debug information). It starts erlang in a distributed mode by default. It includes a slightly modified Eshell which initialises the SSErl environment, and executes any commands given using the modified apply in an sserl environment. Thus all the new BIFs are available.

An alternate script `nsserl` is provided which uses a custom boot file `start_sserl.boot`, which must be generated from an appropriately customised copy of `start_sserl.script` using `mkboot:mkboot(start_sserl)`. This script is much more dependent on the Erlang system structure.

A number of additional utility routines are provided in the `sserl` module, and have been incorporated into `shell_default`, and are available directly from the shell prompt. These are all described in the shell `help()`. Some of the most useful include:

info()
info(Node)
    which display the node status information (rather long and verbose).
ps()
ps(Node)
    which lists all processes executing on a node
names()
names(Node)
    which lists all registered names on a node
mknode(Name)
safenode(Name)
    create a new unrestricted or limited subnode

Subnodes are created by the `newnode(Name)` BIF (or the safety `policynode(Name,Policy_Module)` or `safenode(Name)` library functions, see [Bro97b]). A capability for the new node is returned. This may then be used with `spawn` to run processes in the node.

Some functions are provided in the `test` module in the test subdirectory, to exercise various aspects of the SSErl environment, particularly focusing on the modified BIFs. See `test:help()` for details of the various test functions.

An abbreviated sample sserl session is given in the listing below. It assumes sserl was started in the test subdirectory of the distribution. Some details have been omitted for brevity.

```
UNIX> sserl
Erlang (JAM) emulator version 4.4

Eshell V4.4  (abort with ^G)
SSErl Node 'lpb@galaxy.serc.rmit.edu.au' initialised.
(lpb@galaxy.serc.rmit.edu.au)1> help().
** shell internal commands **
... various standard output omitted
** commands in module sserl (SERCs Safer Erlang) **
init()                -- Create top node (done by shell).
help()                -- Displays this help.
info()                -- Displays info about top node.
info(Cnode)           -- Displays info about node.
... other help details omitted

(lpb@galaxy.serc.rmit.edu.au)2> info().
Node Info Details
  Name           'lpb@galaxy.serc.rmit.edu.au'
  Node Capa      {capanode,'lpb@galaxy.serc.rmit.edu.au',#Ref,7330370}
  Parent Capa    topnode
  Subnodes
  Processes
   {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,2092775} -> <0.27.0>
   {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,2248609} -> <0.21.0>
  Process Cnt    2
  Process Rights [..rights..]
  Capabilities
   {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,2092775} -> {process,<0.27.0>,[..
   {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,5056377} -> {process,<0.15.0>,[..
   {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,2583009} -> {process,<0.20.0>,[..
   {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,2248609} -> {process,<0.21.0>,[..
   {capanode,'lpb@galaxy.serc.rmit.edu.au',#Ref,7330370} -> {node,<0.25.0>,[..ri
  Seed           {667,25635,181}
  Names
   file_server -> {capapid,'lpb@galaxy.serc.rmit.edu.au',#Ref,5056377}, <0.15.0>
  Aliases        []
  Options        []
  Status         alive
  Ticker         <0.26.0>
ok

(lpb@galaxy.serc.rmit.edu.au)3> N1=safenode(saf1).
{capanode,'saf1.lpb@galaxy.serc.rmit.edu.au',#Ref,5591103}

(lpb@galaxy.serc.rmit.edu.au)4> P1=spawn(N1,test,test,[]).
{capapid,'saf1.lpb@galaxy.serc.rmit.edu.au',#Ref,5293704}
Test - simple test to see self - at time {15,55,9}
test: self {capapid,'saf1.lpb@galaxy.serc.rmit.edu.au',#Ref,5293704} -> {process
Process List for node 'saf1.lpb@galaxy.serc.rmit.edu.au'
Capa#       Pid          Current Call
```

```
5293704  <0.32.0>     {sserl_bifs,k_process_info,2}

(lpb@galaxy.serc.rmit.edu.au)5> ps(N1).
{capapid,'saf1.lpb@galaxy.serc.rmit.edu.au',#Ref,7545331}
Process List for node 'saf1.lpb@galaxy.serc.rmit.edu.au'
Capa#       Pid          Current Call
7545331  <0.33.0>     {sserl_bifs,k_process_info,2}
5293704  <0.32.0>     {test,snooze,1}

(lpb@galaxy.serc.rmit.edu.au)6> stop().
UNIX>
```

Programming in the Safe Environment

Programming in the safe environment should be very little different from normal erlang coding, save that some operations may be restricted when the code is executed. Generally the new BIFs would only be used in creating a custom environment, or in some utility modules (which handle display of capabilities for example). As an example, the utility function `safenode(Name)` which is supplied as part of a suite of utility functions in the `safety` module, is listed below (note it uses other utility functions from the `safety` and `ordsets` modules).

```
%% safenode/1 - creates a "safer" subnode of the parent
%%    node rights exclude delete_module,load_module,net_kernel,newnode
%%    processes within it may not use group_leader,open_port,priority
safenode(Name) ->
    CParent = get_dict(node),                   % get parent capability
    % restrict node rights list from parent for new node
    Ri = view_rights(CParent),                  % get rights of parent node
    NR = subtract(Ri,
                 list_to_set([delete_module,load_module,net_kernel,newnode])),
    % restrict proto process rights for new node
    St = node_info(CParent),
    PR = subtract(St#ninfo.p_rights,
                 list_to_set([group_leader,open_port,priority])),
    % add safe module aliases to Aliases table
    NewAli = [{file,safe_file},{lists,safe_lists},{ordsets,safe_ordsets},
                {random,safe_random},{string,safe_string},{unix,safe_unix}],
    Ali = append(NewAli, St#ninfo.aliases),
    % start the safe versions of daemons used by safenode modules
    catch safe_file:start(),
    % create new node with safer rights and custom world-view
    newnode(CParent,Name,{NR,PR,nil,Ali,nil,nil}).
```

This also demonstrates the use of aliases. The `safenode` library function uses the safe_file_server. It restricts file access to the current directory only, but is accessed using the usual `file` functions, with the module name being appropriately aliased.

Limitations of the SSErl Prototype

The current prototype has a number of limitations.

- most of the current library modules are not (yet) compiled in the safe environment. Whilst they should run, they will not see the custom subnode environment, nor will they handle capabilities
- performance will be reduced due to the extra layer of glue functions, and the necessity to exchange messages with the node manager process for all unsafe BIFs
- open_port functionality involves a visible change in use, since the real underlying Pid must be used as part of the communications dialog

- display of capabilities for processes and nodes is obviously different to what is seen at present (though presumably the io_lib functions could be changed to hide this)
- values returned for nodes are currently inconsistent, some functions return a node capability, others return a node name. In part this is due to not having underlying node capabilities in the net_kernel.

All of these limitation could be addressed by incorporating the changes directly in a new version of the Erlang Run-Time System.

Incorporating SSErl in the Erlang RunTime Environment

Once experience with this prototype has verified the validity of this approach to providing a safe erlang execution environment, it would be much better to incorporate the changes into a new version of the ERTS. Also at this time, further safety checks could be made on the manner in which the ERTS has been written.

Capabilities and Subnodes in the ERTS

Capabilities should be a fundamental erlang data type, similar to a reference. It would be uniquely tagged of course, and should include some identifier for the subnode which created the capability (probably an index into the atom table, as I believe is done now for processes and references), along with a random value selected unpredictably and sparsely from a large possible space. This should use around 128 bits. Any pseudo-random generator function used **must** be seeded with information that is hard to predict externally (ie some combination of time and current system data structures at least, a true random source would be ideal though). These capabilities would be used for processes, ports, and nodes, as well as extensibly for other data requiring protection.

Subnodes should be added as a concept in the ERTS. They will primarily involve a table of relevant status information for each distinct subnode, along with some means of locating this table in the system both internally and externally. The table will include much of the information currently managed by the node manager processes in this prototype. The capability table will map capabilities to their underlying values and rights.

The process state information will need to be extended to include capabilities for itself, its parent, and its group leader; and probably a pointer directly to its parent node state table for efficiency. Note the parent node capability need not be the same as that recorded in the node state table, it may very well be a restricted version of it.

All the BIFs implemented in the ERTS which involve potentially unsafe operations will need to be rewritten to incorporate an appropriate check of rights from the supplied (or inherited) capabilities before proceeding.

Auditing BIFs and Standard Library Routines

All BIFs and standard library functions which are written in a general programming language (eg C) will need to be audited for careless coding practises which could be used to subvert the type safety of the system. These have been found to be a major source of security flaws in existing systems (eg see discussion on Java weaknesses in [DFW96]).

This component will be time-consuming, but necessary to ensure safety. Examples of poor style include any use of the standard C functions gets, sprintf, strcat, strcpy; ie any functions which could overrun a buffer supplied to them due to the absence of bounds checks on these parameters. The basic requirement is that all parameters be checked to ensure that bounds are not exceeded, that their values are sane, and cannot cause a run-time execution fault.

Other Changes for Improved Security

Some other changes which could be considered in the ERTS include the placement and implementation of the message buffer, and of the process scheduler.

Currently there is a single message buffer shared by all processes in an Erlang Node. A single buffer was chosen for efficiency and capacity management reasons, but does leave all processes on the Node susceptible to a denial of service attack. This could be created by a rogue process flooding some server with mal-formed messages that are not matched by any receive patterns in that server, and thus not flushed from the buffer. With the implementation of subnodes, consideration should be given to making the message buffer a component of the subnode rather than the node. Also, to reduce the impact of flood attacks, some mechanism for garbage collecting "old" messages, perhaps with a caveat that only messages which have been checked against a pattern and rejected some number of times are eligible. This is similar to solutions proposed to overcome the current spate of TCP SYN attacks on the Internet.

The process scheduler should also probably be modified with the introduction of subnodes. Rather than share CPU cycles amongst all ready processes, consideration could be given to allocating shares to the various subnodes, and then dividing that amongst all processes in a subnode.

Protecting Erlang from External Attack

If the ERTS is assumed to be safe from compromise (ie assume that no-one will gain root type privileges on its host and interfere with its address space(s) directly), then the only mechanism for external subversion is via "spoof" messages being sent to the port(s) associated with the net_kernel in distributed Erlang implementations. At present, the only security mechanism used is to require a suitable "cookie" be sent with each message [AVWW96]. However, this is sent in the clear, and is subject to eavesdropping, and subsequent masquerade by an attacker.

In order to secure these messages being exchanged between distributed Erlang nodes, it is necessary to either physically protect all communications links used, or to employ cryptographic techniques to secure the communications. Possible approaches to the latter involve the use of a "digital signature" instead of a "cookie" (eg perhaps a signed hash using the shared secret), or alternatively, full encryption of all links. The use of SSL (secure socket layer) code would most likely be the best choice here [HY96]. In any case, it would mean that the new "safe distributed erlang" would be incompatible with the existing system. This may, or may not, be a problem.

Conclusions

This paper describes the rationale, design approach, and details of the SSErl prototype of a more secure Erlang execution environment. The prototype will be used to evaluate whether an appropriate level of abstraction has been chosen, and whether the interfaces provided are appropriate for the development of "safe" imported code systems. It is anticipated that once the design approach is validated, it will then be incorporated in a new version of the Erlang RunTime System.

Acknowledgements

The SSErl prototype and this paper were written during my special studies program in 1997, whilst visiting SERC in Melbourne and NTNU in Trondheim, Norway. I’d like to thank my colleagues at these institutions, and at the Ericsson Computer Science Laboratory in Stockholm for their discussions and support.

References

APW86
M. Anderson, R.D. Pose, C.S. Wallace, "A Password Capability System", The Computer Journal, Vol 29, No 1, pp 1-8, 1986.

AVWW96
J. Armstrong, R. Virding, C. Wikstrom, M. Williams, "Concurrent Programming in Erlang", 2nd edn, Prentice Hall, 1996. http://www.ericsson.se/erlang/sure/main/news/book.shtml.

Arms96
J. Armstrong, "Erlang - A Survey of the Language and its Industrial Applications", in *INAP’96 - The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, Hino, Tokyo, Japan, Oct 1996. http://www.ericsson.se/cslab/erlang/publications/inap96.ps.

Bro96b
L. Brown, "Mobile Code Security", in *AUUG 96 and Asia Pacific World Wide Web 2nd Joint Conference*, AUUG, Sept 1996. http://www.adfa.edu.au/~lpb/papers/mcode96.html

Bro97b
L. Brown, "Custom Security Policies in SSErl", Australian Defence Force Academy, Canberra, Australia, Technical Note, Apr 1997.
http://www.adfa.edu.au/~lpb/papers/ssp97/sserl97b.html

DFW96
D. Dean, E.W. Felten, D.S. Wallach, "Java Security: From HotJava to Netscape and Beyond", in *Proceedings IEEE Symposium on Security and Privacy*, IEEE, May 1996.
http://www.cs.princeton.edu/sip/pub/secure96.html.

Erla96
Erlang Systems, "Erlang Distribution", Ericsson Software Technology AB, Erlang Systems, 1996. http://www.ericsson.se/erlang/.

GM95
J. Gosling, H. McGilton, "The Java Language Environment: A White Paper", Sun Microsystems, May 1995. ftp://ftp.javasoft.com/docs/.

HY96
T.J. Hudson, E.A. Young, "SSLeay and SSLapps FAQ", Uni. Queensland, 1996.
http://www.psy.uq.edu.au:8080/~ftp/Crypto/

JNS97
I. Jonsson, G. Naeser, D. Sahlin, et al., "Adapting Erlang for Secure Mobile Agents", in *Practical Applications of Intelligent Agents and Multi-Agents: PAAM’97*, London, UK, Apr 1997. http://www.ericsson.se/cslab/~dan/reports/paam97/final/paam97.ps

LSW95
S. Lucco, O. Sharp, R. Wahbe, "Omniware: A Universal Substrate for Mobile Code", in *Fourth International World Wide Web Conference*, MIT, Dec 1995.
http://www.w3.org/pub/Conferences/WWW4/Papers/165/.

Nae97a
G. Naeser, "Your First Introduction to SafeErlang", CS, Uppsala University, Jan 1997.
http://www.csd.uu.se/~gaffe/general/safe/nae97a.ps.gz

OLW96

J.K. Ousterhout, J.Y. Levy, B.B. Welch, "The Safe-Tcl Security Model", Sun Microsystems Laboratories, Mountain View, CA 94043-1100, USA, Nov 1996. http://www.sunlabs.com/research/tcl/safeTcl.ps.

Tar95

J. Tardo, "An Introduction to Safety and Security in Telescript", General Magic Inc., 1995. http://cnn.genmagic.com/Telescript/TDE/security.html.

Wiks94

C. Wikstrom, "Distributed Programming in Erlang", in *PASCO'94 - First International Symposium on Parallel Symbolic Computation*, Sep 1994. http://www.ericsson.se/cslab/erlang/publications/dist-erlang.ps.

Now Superceeded by SSErl - Prototype of a Safer Erlang